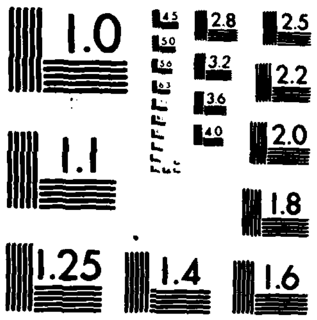
AD-A087 385 DEFENCE AND CIVIL INST OF ENVIRONMENTAL MEDICINE DOW--ETC F/G 5/10
USER EXPERIENCE WITH MODULA FOR PROGRAMMING A REAL-TIME APPLICA--ETC(U)
1979 S M MCFADDEN
UNCLASSIFIED DCIEM-RP-80-P-05 NL

END
DATE FILMED
9 80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
45
50
56
63

(3)

(6)

USER EXPERIENCE WITH MODULA FOR PROGRAMMING A REAL-TIME APPLICATION

(12) 6

(10) Sharon M. McFadden
Defence and Civil Institute of Environmental Medicine
Box 2000, Downsview, Ontario, Canada M3M 3B9

(14) DCIEM-RP-80-P-05 (11) 1979

ABSTRACT

This paper reports on the usefulness of Modula for programming the control of experiments in a psychoacoustic laboratory. Modula was designed for programming dedicated computer systems. As such, it provides facilities for multiprogramming and for the control of machine dependent devices. Using Modula a non-system programmer (the author) was able to write the software for a stand-alone system to control the operation of a number of concurrent activities in an experimental laboratory. This was due to the simplicity of the language, the concept of encapsulating independent activities in to separate "modules" and the use of "signals" to communicate among asynchronous "processes".

INTRODUCTION

A computer controlled laboratory is essential for research in modern psychoacoustics. Precise timing of complex waveforms is required, and the waveforms themselves must be synthesised to exact specifications. The psychoacoustic laboratory at DCIEM has evolved from an analogue network of oscillators, filters, attenuators, noise sources, etc., in this direction. In 1979, a PDP-8I which had been used as a signal generator and controller was retired in favour of a PDP-11/40, which was to be used to create and present the signals, communicate with up to three separate listeners, and control all facets of the experiment.

Using the PDP-8I, assembly-language programming appeared to be the only effective method of performing the required tasks at an adequate speed. The overheads, and more important, the possible timing uncertainties associated with an operating system, made it almost imperative to work with the bare machine. A PDP-11/40 is, for some purposes, a slower machine than a PDP-8I, and speed of execution as well as timing precision are more critical in the more complex system. The complexity of the system and the absence of an experienced programmer precluded writing in PDP-11 assembler code. On the other hand, mono-processing languages like Fortran were both cumbersome as languages and unsuited for development of highly asynchronous concurrent systems. Another consideration in selection of a development methodology was that the 11/40 should not be the development machine. The host machine was an 11/34 running the UNIX time-sharing system, which has a wealth of software development tools, and an early decision was made to take advantage of those tools.

The language MODULA was designed by N.Wirth (the designer of PASCAL), at ETH (Eidgenoessische Technische Hochschule, Zurich) and is described in three consecutive articles in "Software-Practice and Experience" (2, 3, 4). It has been implemented as a cross-compiler running under UNIX for all types of PDP-11 from the LSI-11 to the largest PDP-11s, by the University of York (U.K.) from whom it is available, with maintenance and update services, for a nominal fee[2]. This implementation was chosen for the development of the software for the DCIEM psychoacoustic laboratory, partly as an experiment to see how easy it would be for a researcher with no system programming experience to develop a complicated concurrent real-time system. The results have been pleasing.

THE REAL-TIME APPLICATION

The application for which Modula was chosen was a system to control experiments in a psychoacoustic laboratory. In setting up this laboratory the aim was to provide a facility which would allow the presentation of a wide range of auditory stimuli and the implementation of many different experimental paradigms. In general, our interest is the processing of auditory information including speech and non-speech stimuli. Specific areas of interest are the relative advantages of presenting information auditorily rather than visually, determining methods of improving a person's capability to discriminate signals in noise, and finding cues used by humans in discriminating amongst complex sounds. A common paradigm is to present a stimulus or series of stimuli to an observer and have him or her report on the presence of the signal or of a difference

****

[1] This is DCIEM Research Paper No. 80-P-05 √

****

[2] The address is I. D. Cottam, Department of Computer Science, University of York, Heslington, York, YO1 5DD, England.

406986 [illegible]

amongst the signals.

This research project is a continuing one. Originally, the laboratory for these experiments consisted of a conglomeration of digital and analog components which the experimenter plugged together into whatever configuration was required for the experiment being carried out at the time. Stimuli were modified by twiddling knobs and dials, and data collection meant writing down totals from counters. This kind of system is prone to human error and is not suitable for answering the kinds of questions that now concern us. Over time, parts of the system were put under computer control, namely, a PDP-8I. With the 8I it was possible to generate simple signals and collect and analyse observer responses. Even so, the system still was inadequate to carry out the required research. Therefore, it was completely revamped into the form shown in Figure 1, with the 8I being replaced by an 11-40. In the design of the new laboratory, it was anticipated that signal waveforms would be generated off-line and passed to the 11/40 by a high-speed link from a host computer (PDP-11/34) for buffering via RK05 discs. When wanted, the waveform would be stripped off the discs and passed through an digital to analogue (d/a) converter to the audio system. Communications between the observers and the computer would be via standard terminals for maximum flexibility. Some of the analog equipment used previously would still be required, but it would all be controlled by the 11/40 rather than manually as before.

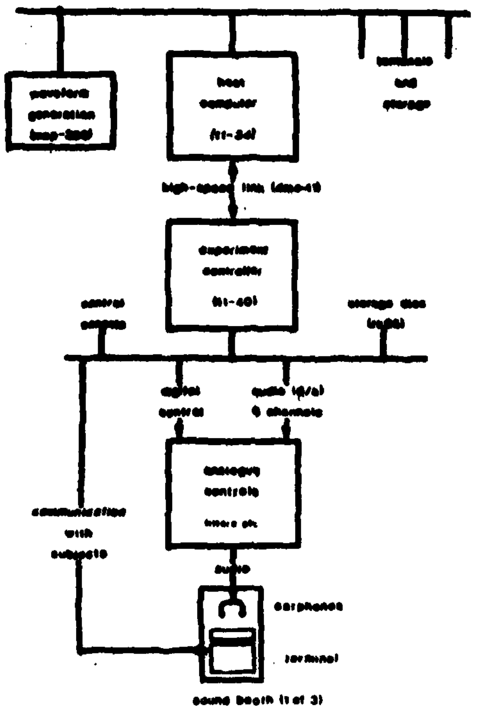


FIGURE 1: A schematic of the hardware in the psychoacoustic laboratory.

With the change in hardware there was a requirement for new software. Since the system is entirely computer controlled, the amount of programming required and the complexity of the programs to be written is much greater than before. In a typical experiment, the computer must control a number of activities at the same time, including the transfer of signals either from a disc or from the host computer to the 11-40 and from there to the observer via a d/a, the presentation of visual timing cues to the observer on a terminal, the attenuation of stimuli, the collection of responses made, and data storage. Up to three observers participate in an experiment at one time and these activities must be carried out for each observer simultaneously and independently. The software for such a system is easy neither to design nor to write. The system supports a number of special purpose peripheral devices to carry out the required activites. Standard handlers are frequently not available for such devices and must be written. Even where standard device handlers exist they often cannot be used. Standard handlers are designed to be all things to all people at the expense of speed of operation. Therefore, they are usually inefficient for this system where many actions involving many different peripherals must be carried out in a brief time frame.

In summary, the application was a system to control the operation of an experimental laboratory with a large number of hardware peripherals. The experimental protocol usually involved controlling a large number of virtually simultaneous events. Moreover, since the protocol varies considerably from experiment to experiment the software must be easily modified.

INITIAL REASONS FOR SELECTION OF MODULA

Given the complexity of the system being developed it was desireable that the software be written in a higher level language. This was doubly important since all of the programming was to be done by a person whose main interest was research not computer systems programming. Not only does this mean a lack of expertise, it also means that the system would be developed piecemeal as time permitted. Therefore, a language was needed that would be simple to learn and use, as well as meeting the requirements of the application. In addition, the plan was to write the software on a PDP 11-34 under the UNIX operating system in order to give the programmer access to the software tools available under UNIX and the hardware associated with a large timeshare system (printers, magtapes, large discs etc.). Therfore, the language used had to be available under UNIX. The alternatives were to develop a system using the RT-11 operating system or to write a stand-alone system using either C or Modula. Using RT-11 meant learning another operating system as well as a language in which to write the special purpose handlers and experiments. There was also a distinct possibility that existing handlers would have to be rewritten. C provided the capability for writing a stand-alone system. However, Wirth's article on Modula (2) suggested it was more suitable for this kind of application.

As stated previously, Modula was designed for programming a dedicated computer system such as those used in process control systems and experimental laboratories. As such, it has facilities for controlling the operation of peripheral devices and it is claimed to produce highly efficient code. Modula is based on Pascal and has many of the constructs found in that language. The compiler developed at York University in England for use under UNIX makes it possible to write a stand-alone system for a PDP 11. The large test plan and the real-time application programs included with the compiler, simplified the introduction to the language.

USEFULNESS OF MODULA

Modula more then met initial expectations. It proved extremely simple to learn to use and to write the software for the system. Even though the software was written over a span of many months with much of that time spent on other projects (including one period of almost two months), the time spent in relearning was essentially nil. Even the most complex experiment was straightforward to program because of the capability for expressing concurrency that Modula provides. Writing device handlers was no more difficult than programming any other function.

A detailed description of the language is not possible here nor would it serve any real purpose. Wirth provides an excellent explanation of Modula and how it is used in the three articles mentioned in the introduction (2, 3, 4). An appreciation of the usefulness of Modula is better achieved by a brief description of some of the major constructs in terms of how they were used in writing the software for this system.

<u>The Module</u>

A program usually consists of a number of "modules" each dedicated to a specific function such as I/O, data storage, signal generation etc. A module is a collection of procedures, variables and constants which are segregated from the rest of the programming environment. They are declared local to the module and are not known to procedures outside it unless their identifiers are deliberately exported by being included in a "define" list. Similarly, any variables or routines defined outside a module are not known inside it unless they are included in the module's "use" list. However, procedures and processes declared within a given module have access to all variables declared local to that module. In this way it is possible to have a number of subroutines (in Modula called procedures) which have access to common variables, yet still restrict the scope ,and visibility of those variables. This minimizes interactions and simplifies coding of very large programs.

Modules can be nested within modules making it possible to build functional units composed of two or more modules. Variables are shared amongst the sub modules but are not visible to the remainder of the program. This makes it possible to write a program as a collection of small relatively independent units. In fact the program itself is a module identical in form to all other modules. A module may in one application be the total program and in another just perform one of a number of actions being caused by another program.

In writing up this system considerable advantage was taken of the module. Instead of having to consider the design of a complex program to run a particular experiment it was only necessary to consider the design of a particular functional unit to do a specific task. Initially, a number of basic modules which carried out specific functions, such as device handlers, were written. These basic modules were then combined into a functional unit that conformed to a typical task that had to be carried out. This function was added to a library of functions which could be combined to control any number of different experiments with very little additional effort.

<u>Expressing Concurrency - The Process and Signals</u>

At run time as well as at design time, each task is an independent functional unit because of the capability Modula provides for expressing concurrency. This makes designing the system as a series of functional units even more reasonable. At run time tasks are each started in turn and then appear to run concurrently with one another, communicating by means of shared variables and "signals". Separate concurrent routines are called "processes". A process is similar in form to a procedure except that it runs concurrently with the routine that called it. It can both send and wait for signals, and test whether anyone is waiting for a particular signal. If the end of a process were reached it would not return to the calling routine but simply go out of existence. To prevent this (a fatal error), a process contains a loop statement without an exit condition (see Figure 2) so that once it is initiated it continues indefinitely. However, it is not executed continuously. Most of the time all but one process will be waiting for control to be passed to it. A program usually consists of a number of processes with control being passed from one to another as events dictate. Each process carries out a different task. To the user the tasks appear to be going on simultaneously.

Control is passed from process to process via signals. A signal is like a variable except that it can not have a value. In fact only two operations, the procedures "wait(s,n)" and "send(s)", and one test, the procedure "awaited(s)" can be applied to a signal. The procedure wait(s,n) delays at priority "n" further execution of the process until the signal "s" is sent by some other process. Send(s) passes control to a procedure that has been waiting for "s" the longest or with the highest priority. Awaited(s) returns true if at least one routine is waiting for "s". When a wait is encountered, control passes to some other process being executed concurrently. If a send is executed, control passes from the routine with the send to a routine waiting for the signal. Control returns to the sending routine only after the receiving routine finishes execution of its code or another wait is encountered. This linkage of shifts of control at "wait" and "send" statements means that Modula is well suited only to mono-processor applications. In a multiprocessor environment, a process that "send"s a signal would not need to wait while the concurrent one began execution.

Part of the code for a process which controls the timing of events in a trial is shown in Figure 2. A trial is broken up into five parts - get

```
PROCESS trialtime;
USE setclk, clkoff, timtab, runon, runstrt,
    si1, si2, siover, resptime, ready;
BEGIN
 LOOP
  IF runon THEN
   setclk(timtab[0]);
   WHILE awaited(ready) DO send(ready) END;
   wait(clkoff); setclk(timtab[1]);
   WHILE awaited(si1) DO send(si1) END;
   wait(clkoff); setclk(timtab[2]);
   WHILE awaited(siover) DO send(siover) END;
   setclk(timtab[1]); wait(clkoff);
   WHILE awaited(si2) DO send(si2) END;
   . (* etc *)
   setclk(timtab[3]); resptime:=true;
   wait(clkoff); resptime:=false;
  ELSE wait(runstrt,obs) END
 END
END trialtime;
```

FIGURE 2: An example of the code for a process. Other processes and procedures would wait for the signals "si1", "si2", and "siover".

```
PROCESS playsig(obs:integer);
USE si1, si2, random, sigon,
    sigoff, runon, runstrt;
VAR a:integer;
BEGIN
 LOOP
  IF runon THEN
   a:=random(2);
   IF a=1 THEN wait(si1,obs)
   ELSE wait(si2,obs) END;
   sigon(obs); wait(siover,obs);
   sigoff(obs);
   IF resptime=false THEN wait(siover,obs) END;
  ELSE wait(runstrt,obs) END;
 END
END playsig;

PROCESS response(obs:integer);
USE resptime, read, runstrt, runon,
    answer;
VAR mess:char;
BEGIN
 LOOP
  IF runon THEN
   read(obs,mess);
   IF resptime THEN
    answer[obs]:=mess END;
  ELSE wait(runstrt,obs) END
 END
END response;
```

FIGURE 3: Two processes that could run concurrently with trialtime. Playsig plays a signal in either stimulus interval one or two. Response waits for the observers response and stores it in "mess".

ready, stimulus interval one, interstimulus interval, stimulus interval 2, response interval. The signal is presented in one of the two stimulus intervals. As can be seen this process sets up a clock interrupt to happen some time in the future, waits for it, then executes the appropriate code related to that moment in time. This often includes sending signals to other processes or setting flags for other processes to check. Figure 3 shows two processes that would run concurrently with trialtime. Communication between these processes would be by means of the signals "si1", "si2", and "siover" and the boolean variable "resptime". Playsig waits for "si1" and "si2" and "siover". When it receives either "si1" or "si2" it starts an auditory stimulus playing. When it receives "siover" it stops the same stimulus. Response uses "resptime" to determine if the message being sent by an observer was sent during the response interval. It is possible to run more than one subject, by starting each process in Figure 3 more than once. Each time an instance of a process is started the value of "obs" (the parameter each process is called with) would be different. "Runon" and "runstrt" are used to activate and inhibit these processes at the start and finish of each run.

Peripheral Devices

Providing the capability to program the handlers for peripheral devices in a higher level language is usually difficult because, as Wirth states

> "such facilities are inherently machine- and even configuration-dependent and as such elude a comprehensive abstract definition."

Wirth's solution was to enclose the machine dependent items inside a special type of module called a device module (see Figure 4). The features of the device module reflect the features of the machine the compiler was written for - in this case a PDP-11. At the same time they are practically identical in form and use to standard Modula constructs. There is very little extra to learn in order to write a handler in Modula. The extra features include a means of establishing the priority at which a device operates, a means of providing access to hardware registers and a means of handling interrupts.

The integer number at the end of the DEVICE MODULE declaration is that device's priority. All processes and procedures within the device module operate at this priority. This means that while code is being executed within a device module all hardware or software interrupts of equal or lower priority are shelved until control leaves the high priority module. Consequently, any operations on a device's registers can be carried out at the same priority as the relevant interrupt.

The hardware registers associated with a device are also treated in a straight-forward manner. Each is defined as a variable like any other variable, but with the hardware address written in brackets following the identifier (for example, the declaration of "lcs[177546B] : bits" in Fig 4). The register may be treated like any other variable and the most suitable variable type can be used. Usually, the status register is declared to be of type "bits" (an array of 16 boolean variables) while buffer

```
DEVICE MODULE clock[6];
DEFINE tick, time;
VAR time:integer;
    tick:signal;
    lcs[177546B]:bits;
(* In Modula, octal constants
are terminated with the character B *)

PROCESS kw11clock [100B];
VAR ticks:integer;

BEGIN
  ticks:=0;
  LOOP
    lcs[6]:=true;
    doio;
    inc(ticks);
    IF ticks=60 THEN inc(time); ticks:=0 END;
    send(tick);
  END
END kw11clock;

BEGIN
  time:=0; kw11clock
END clock;
```

FIGURE 4: A device module to control the operation of the kw11-1 clock. This module only contains the process to handle the interrupt. Other procedures that alter the value of "lcs" could also be included. Outside the module clock "time" is read-only and "tick" can only be waited for. "Ticks", the tick count, is invisible outside the process kw11clock.

registers are declared to be type "integer" or "character". In this way, individual bits in the status register can be modified at will, while the contents of the buffer can be dealt with as the numbers or characters that they truly represent.

The actual hardware interrupts are handled in a "device process", which functions in the same way as any other process except that it will ordinarily contain a "doio" instruction. Only a device process can contain a "doio" statement and only one instance of a device process can be activated at a time. Each device process declaration includes the address of the interrupt vector (for example, the line "PROCESS kw11clock [100B];" in Fig 4). The doio statement is the link between the actual interrupt and the user program. It acts like a wait statement, for which the signal is sent by the system when an interrupt happens at the vector specified in the device process declaration. When a doio is executed, control is relinquished by the device process until the proper interrupt occurs. At that time, control returns to the device process just as in any ordinary interrupt handler, unless a higher priority process is in execution. Normally, the device process will disarm the interrupt, do its work, rearm the interrupt, and loop back to the doio statement to await the next occurrence of its interrupt.

The extra facilities of device modules are similar in form and use to the rest of the Modula constructs. Each device handler can be written as a separate unit. Its internal details, including the operation of the device process, are invisible to the rest of the system.

Writing a device handler offered no special problems, even though the writer had no previous experience with device handlers. The structure that had to be followed was clear and concise. It was easy to translate the specifications in the hardware manual into a device module. The simplicity of structure also meant that the scope for error was less which cut down the time required for debugging a routine. Integration of several device handlers was never a problem. It is taken care of by specifying a priority and by the use of a device process. At run time the device process for each device is initiated. Thereafter, whenever the interrupt for a particular device is enabled, interrupts to that device are handled at the priority specified. Devices operate concurrently with one another in conjunction with ordinary processes.

The primary reason for writing special purpose handlers was the inefficiency of standard handlers. The Modula compiler is written with code efficiency as a major design goal. The test of this was the signal playing function. Successive samples of a waveform are transfered from a buffer in core to the d/a, one word at a time, on successive interrupts of a programmable clock which runs at top priority. This buffer is refilled when required, from an RK05 disc, double buffered. If the clock interrupts happen at too high a rate nothing else gets done. The Modula system can handle the disc-to-core transfer, the core-to-d/a transfer, the clock interrupts, and terminal interaction with a clock interrupt once every 90 microseconds. It is doubtful if it would be possible to achieve a much better rate of transfer under any other system.

In summary, as claimed (2), the language was well suited for this kind of application. Not only is it possible to write the concurrent software required, but as an additional benefit, Modula almost seems to force one to write more clean, elegant and generally useful software. Writing functional units that do a specific task is potentially more useful and more flexible than writing a total system right from the start. Defining the actions as separate processes which all run relatively concurrently results in a program which corresponds more directly to the actual operation of the experiment. This makes it easier to understand the software and hence to debug it and to modify it later on.

LIMITATIONS OF MODULA

The limitations of Modula arise mainly out of one of the features that initially attracted us to it; namely, it was simple to learn. Modula was designed to be as simple as possible so that it could run on very small computers. Wirth's philosophy of language design is to provide as few primitives as possible for the job, to ensure that these are defined in a natural and general way, and to strip off all possible "ornamentation". Consequently, a number of potentially very useful constructs available in most higher level languages do not exist in Modula. There is no capability for indirect (pointer) addressing, no inherent file structure, no standard I/O commands, no data type "float" ("real"), and hence no floating-point arithmetic. The absence of I/O commands and of a file structure is required by the philosophy of Modula. It is not, and does not replace, a generalized operating system. It was designed for the construction of process-control applications, and the effective description of I/O processes in a concurrent asynchronous environment. The absence of indirect addressing is required by the strong segregation of modules. Indirect addressing could permit modules to affect other modules if the pointers were incorrectly set, resulting in bugs of great difficulty in a complex concurrent system. The absence of indirect addressing is, however, annoying when dealing with devices such as a DMC-11 (a high-speed inter-computer link), which asks for and returns the address of a buffer. It would save time and storage to be able to store that address in a pointer and pick up the contents of the address by indirect addressing. Floating-point arithmetic is also very useful in many real-time applications. However, it is also very time consuming if there is no floating point hardware on the machine. Therefore in many applications, such as this one, it could not be used whether it was available or not. Nevertheless, provision of a "float" data type would probably be one of the most useful extensions of Modula.

The small number of constructs limit the range of applications for which Modula is useful. It is not a general purpose language. This was noted by Holden and Wand (1) in their assesment of Modula. They felt that Modula was most useful for the development of small real-time systems written by a single programmer. It would not be possible to write a general operating system in Modula because it does not allow pre-emption which is necessary to implement a scheduler. Since what it does, it does well, it is worth learning Modula to write the software for a small real-time system.

THE COMPILER

The amount of use a language gets on a particular system usually depends on the quality of the compiler written for that system. The implementation used in this application was Release 2.0 of the University of York (UK) Modula compiler. This compiler is written in BCPL, and can be built on a PDP-11 running UNIX, or on a DEC-10 or DEC-20. It runs under UNIX, and produces code for any bare PDP-11, from an LSI-11 to an 11/70. Normally, the target machine will be one of the smaller PDP-11's. There is a possibility that versions for DEC operating systems on the PDP-11 will be available. Release 3.0 is expected to be available on UNIX in Spring 1980. The PASCAL User's Group Newsletter lists other implementations of Modula, with various target machines, and running on various hosts.

Our experience with the York compiler has been mainly favourable. To start with, it was easy to install. It compiled and ran on the first attempt. This included compilation of all the BCPL source code using a binary-only BCPL compiler supplied with the Modula system. The compiler includes a large test plan (about 150 pages of Modula code) to check out each of the constructs of the language. This has proved of inestimable value. It shows clearly by example how to use each construct and what happens when you do; it provides many examples of ways to code interesting functions and processes; and it proves that the compiler is working reasonably reliably. This last feature is a great psychological aid when program bugs appear. You can believe that the bugs are your own fault rather than the fault of the compiler. This belief has so far been well-founded. There are a few known compiler bugs, for which fixes are available from the University of York as part of the maintenance service, but these bugs seldom appear. When they do, it is usually obvious that there is a compiler bug rather than a program bug.

CONCLUSION

Writing a stand-alone system for an experimental laboratory is not a simple task, no matter what the language. The requirements are stringent and usually the work is never completed, as the uses for the system are constantly shifting. With Modula the task seemed more reasonable. It was written for this kind of application. Consequently, one does not have the problem of adapting constructs to fit one's needs. Rather the constructs seem to assist in the definition and design of the system. The result is that it is possible for a non-system programmer to design, write and debug a stand-alone system and produce software that is easy to read, flexible, efficient, and easy to modify at a later date.

REFERENCES

(1) Holden, J., and I. C. Wand, An Assessment of Modula. York Computer Science Report No. 16, University of York, Heslington, England, 1978.

(2) Wirth, N., Modula: A Language for Modular Multiprogramming. Software-Practice and Experience, 7, 3-35, 1977.

(3) Wirth, N., The Use of Modula. Software-Practice and Experience, 7, 37-65, 1977.

(4) Wirth, N., Design and Implementation of Modula. Software-Practice and Experience, 7, 67-84, 1977.

DA

FIL

9 -